

Segmento di somma massima

corso di laurea in **Informatica**

Progettazione di Algoritmi, **Ivano Salvo**

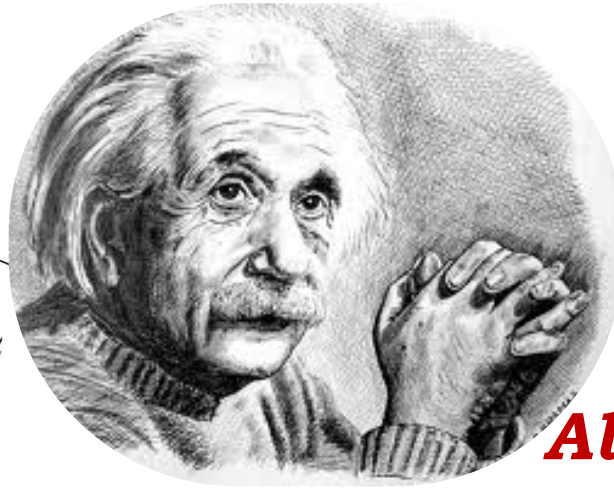
Esercitazione **8** [5/5/26]



SAPIENZA
UNIVERSITÀ DI ROMA

C'erano una volta...

***Henry,
un ingegnere***



***Albert,
un fisico***



***Penelope,
un'informatica***



***Sherlock,
un logico***

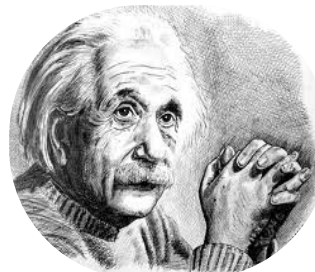
... alle prese con un problema ...

... di grafica che necessitava di calcolare molte volte il **segmento** (= sequenza di elementi adiacenti) di **somma massima** di un vettore.

Il problema è ovviamente **interessante** quando il **vettore contiene sia numeri negativi che positivi**: se contenesse solo positivi, la soluzione sarebbe banalmente tutto il vettore, mentre se contenesse solo negativi, sarebbe il vettore vuoto di somma 0 (o il massimo, se non ammettiamo segmenti vuoti).

Formalmente, **occorre determinare due indici** *inf* e *sup* tali che:

$$\forall i \leq j \in [0, n). S(v[inf, sup]) \geq S(v[i, j])$$



Soluzioni buone per tutte le stagioni

Comincia **ovviamente** l'ingegnere, che scrive un programma in pochi secondi, applicando pedissequamente le definizioni del problema, trasformandolo in un programma e lo mostra gongolante.

“È ovviamente sufficiente **esplorare tutte le coppie** di indici $i < j$, calcolare $S(v[i, j])$ e **selezionare il massimo**”, commenta.

Siccome è **fautore del riuso** delle **componenti software**, si ricorda di una vecchia funzione `somma(v, inf, sup)`, che calcola la somma di un segmento di vettore tra gli indici *inf* e *sup*.

```
def segMentoMax(v):  
    max, n = 0, len(v)  
    for i=0 to n-1:  
        for j=i+1 to n:  
            s = somma(v, i, j)  
            if s > max: max = s  
    return max
```

$\theta(n^3)$

$\theta(n^2)$
coppie i, j

$\theta(j - i)$,
 $n/2$ in media



Argute piccole ottimizzazioni

“Ma siccome è sempre vero che $S(v[i, j+1]) = S(v[i, j]) + v[j]$ ”, osserva il fisico, “non vedo la necessità di tanta complicazione, Henry”

E continua: “anche volendo esplorare tutte le coppie di indici, **non c'è motivo di ricalcolare ogni volta le somme**, ma si possono fare incrementalmente, con gran risparmio”.

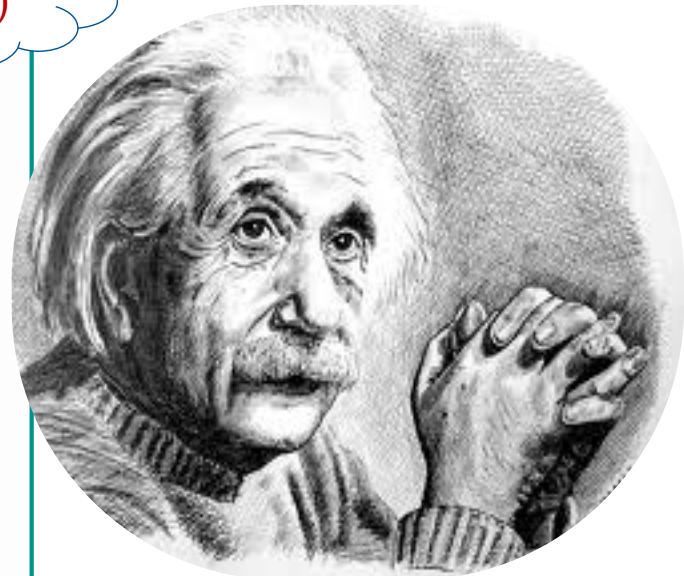
Questo purtroppo rende impossibile il riuso di somma, ma necessita di calcolare la **somma mentre si esplorano i segmenti**.

```
def segmentoMax(v):  
    n = len(v)  
    max, b, e = 0, 0, 0  
    for i=0 to n-1:  
        s = 0 -- calcolo tutte  $S(v[i, j])$   
        for j=i to n-1:  
            # INV  $s = S(v[i, j])$   
            s = s + v[j]  
            if s > max: max = s  
    return max
```

$\theta(n^2)$

$\theta(n^2)$
cicli

$\theta(1)$

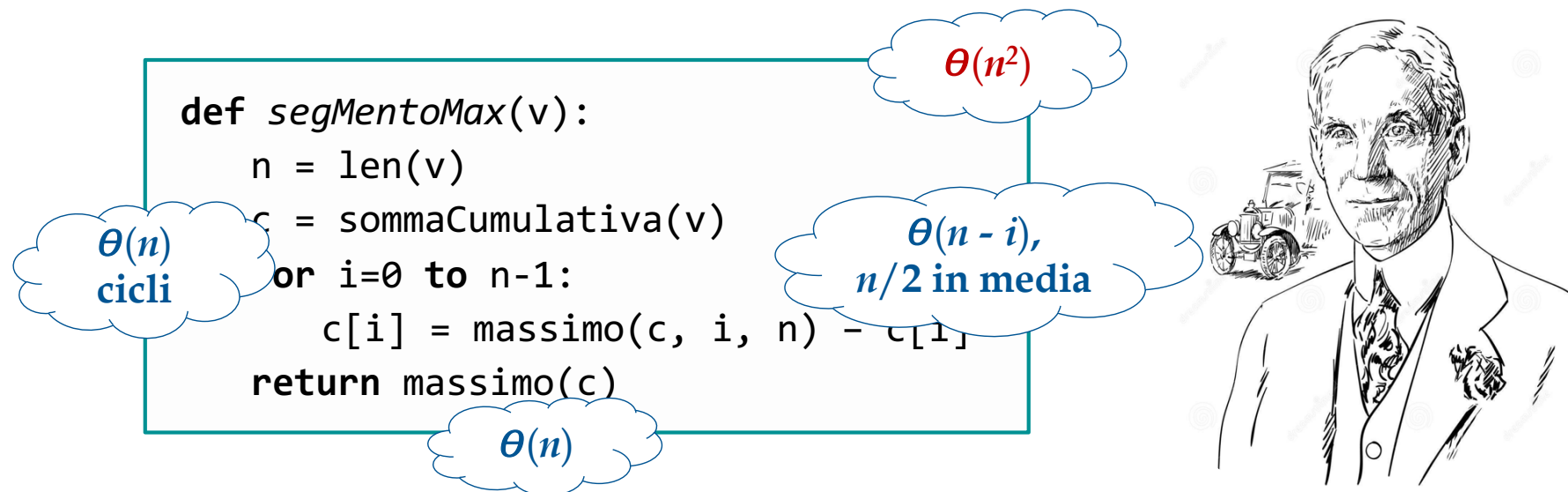


Composizionalità, uber alles

“Caro Albert, si può fare in $\theta(n^2)$ anche essendo **ancora più composizionali**. Ho trovato una vecchia funzione che calcola la **somma cumulata** di un vettore. Dato un vettore v lungo n , restituisce un vettore c lungo $n+1$ tale che: $\forall i \in [0, n). c[i] = S(v[0, i])$.

Con il vettore c , ovviamente $S(v[i, j]) = c[j] - c[i]$ ($i \leq j$).

In realtà, $c[i] = \max(c, i+1, n) - c[i]$ calcola il **massimo dei segmenti con estremo sinistro i** , lo carica in c al posto i ($c[i]$ non servirà più) e dopo è sufficiente fare il massimo sul vettore c ... un **Lego sublime**...”



La parola all'esperta di dominio

Penelope lascia fare, confidente che la sua esperienza sul tema le permetterà di surclassare gli stimati colleghi, **algoritmisti per caso**.

Dopo aver visto la terza soluzione, estrae dalla faretra una delle sue frecce più acuminata: il **divide et impera**.

“Il problema **non è** - a dire il vero - **molto semplice**, perché dividendo il vettore in due, certamente il segmento di somma massima è maggiore o uguale del segmento di **somma massima della parte sinistra** e di quello della **parte destra**, ma occorre tener conto di eventuali **segmenti ottimi a cavallo tra le due metà**.”

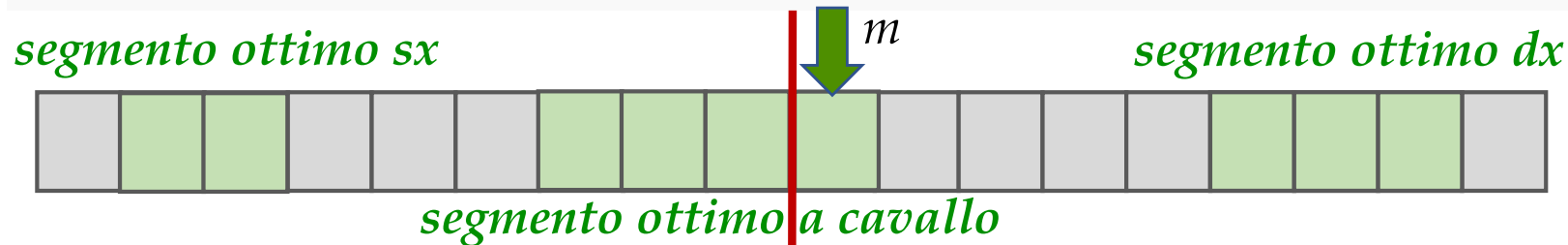
“Ora devo andare, ma confido domattina di portarvi una soluzione migliore di quella quadratica”



Una procedura divide et impera

“Pensavo fosse più difficile” comincia, con altezzosa superiorità. “Il fatto è che il segmento di somma massima, **fissato un bordo**, è molto facile: basta sommare da sinistra a destra (o da destra a sinistra) e **registrare il valore massimo incontrato**”.

“E necessariamente, un eventuale **segmento ottimo a cavallo** di due metà è la *giustapposizione* del **segmento ottimo** attaccato al **bordo destro** della **metà di sinistra**, e del segmento ottimo attaccato al **bordo sinistro** della **metà di destra**... ok, vi faccio il **disegnino**.”



```
def segmentoMaxBordo(v, a, b, p):  
    max, s = 0, 0  
    for i=a to b step p:  
        s = s + v[i]  
        if s > max: max = s  
    return max
```

generalizzo il **passo p**
per andare sia in
avanti che all'indietro

$\Theta(n)$,
 $n=b-a$

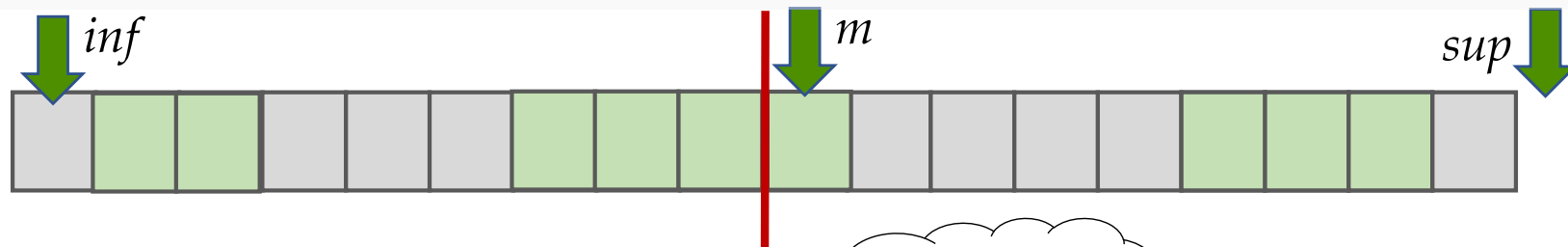


Finalmente il codice e l'analisi

“E voilà, bello no?”

“E l'analisi di complessità conduce a una equazione di ricorrenza ben nota, **la stessa di mergeSort**, che vi ho spiegato l'altro giorno.

$T(n) = 2T(n/2) + \theta(n)$ $T(1) = \theta(1)$, e il risultato $\theta(n \log n)$.



```
def segMentoMax(v, inf, sup):  
    if inf==sup: return 0  
    m = puntoMedio(inf, sup)  
    maxC = segmentoMaxBordo(m-1, inf, -1)  
           + segmentoMaxBordo(m, sup, 1)  
    maxL = segMentoMax(v, inf, m)  
    maxR = segMentoMax(v, m, sup)  
    return max(maxC, max(maxL, maxR))
```

$\theta(n \log n)$

$\theta(n)$

$T(n/2)$

$T(n/2)$



Dedurre le peculiarità del problema

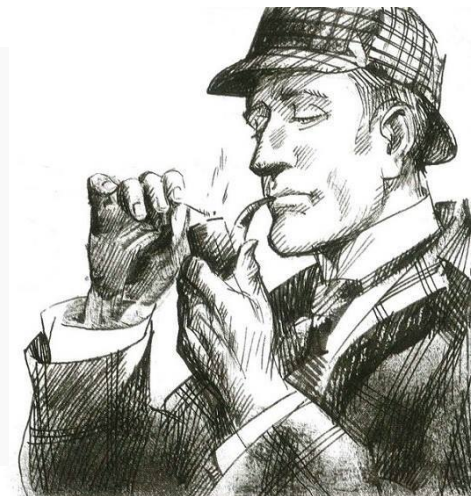
“Interessante, Penelope. Ma anch’io, stamattina all’alba, al risveglio, ho giocato un po’ con i segmenti” osserva il Sherlock Holmes.

“Da principio, ho pensato che un **segmento ottimo** **comincia e termina** necessariamente con un **positivo**, altrimenti potrei togliere il primo o l’ultimo elemento”.

“Ho pensato poi che anche che gli **elementi adiacenti esterni** al segmento ottimo (prima e dopo) **sono necessariamente negativi**, altrimenti **potrei prolungare il segmento** in uno di somma maggiore, **bordi a parte**, s’intende”.

“Questi argomenti si estendono subito: **tutti i segmenti adiacenti esterni** devono **avere somma negativa**”

Dentro il segmento ottimo, viceversa, **tutti i prefissi** e tutti i **suffissi** di un **segmento ottimo** devono avere **somma positiva**.”

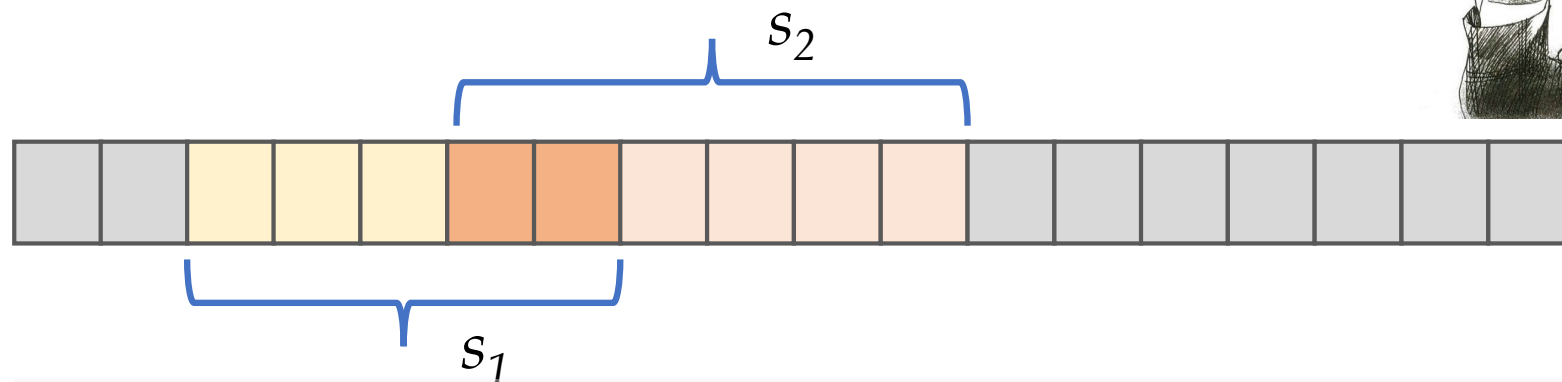


Proofs by diagrams

“È anche chiaro, che in un vettore ci possono essere più segmenti non prolungabili in segmenti di somma maggiore. Per questo io li chiamo **massimali**. E converrete con me che il **segmento di somma massima** deve essere **massimale**”.

I tre amici ascoltano attoniti e hanno già capito che ci sarà un finale a sorpresa, ma ancora non vedono la soluzione.

“Infine, i **segmenti massimali** devono essere **disgiunti**”



“Assumendo s_1 , s_2 massimali: prefisso giallo di s_1 ha somma positiva, ergo posso prolungare s_2 contro l’ipotesi di massimalità. Lo stesso per il suffisso rosa di s_2 .”

Elementare Penelope... anzi lineare

“Un **segmento massimale**, quindi, comincia o **da un bordo**, o dopo **un'area di somma negativa**. Finché la somma è positiva è potenzialmente il **prefisso di un segmento massimale** più lungo”.

“Quando la somma diventa negativa, comincio a vedere se c'è un nuovo segmento massimale”.

“Infine, **registrando i massimi incontrati**, esattamente come in **segmentoMaxBordo** di Penelope, non mi devo preoccupare di tagliare eventuali suffissi negativi”. “Banalmente lineare, amici.”

```
def segmentoMax(v):  
    max, s, n = 0, 0, len(v)  
    for i=0 to n-1:  
        #INV: s è somma di un prefisso di  
        # un segmento potenz. massimale  
        s = s + v[i]  
        if s > max: max = s  
        if s < 0: s = 0  
    return max
```

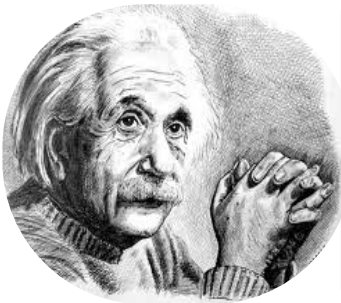
$\theta(n)$



Morali della favola



La **forza bruta** è sempre un'opzione per scrivere un programma corretto in poco tempo (**rapid prototyping**). Questo è aiutato dal **riuso di componenti software** già scritte e testate.



La complessità degli algoritmi e l'efficienza dei programmi possono **dipendere da piccoli accorgimenti**, ad esempio evitare di ricalcolare più volte le stesse cose.

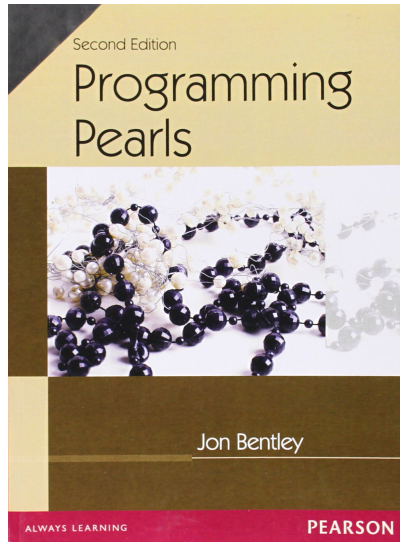
Le ricette pre-confezionate e le **analogie** con soluzioni di **problemi noti** sono sempre una valida guida e vanno considerate.



Ma su tutto, vince sempre l'analisi accurata delle peculiarità del problema in esame.



La vera storia...



[John Bentley, *Principle of Algorithm Design*,
Communication of ACM, 1984]

Principles

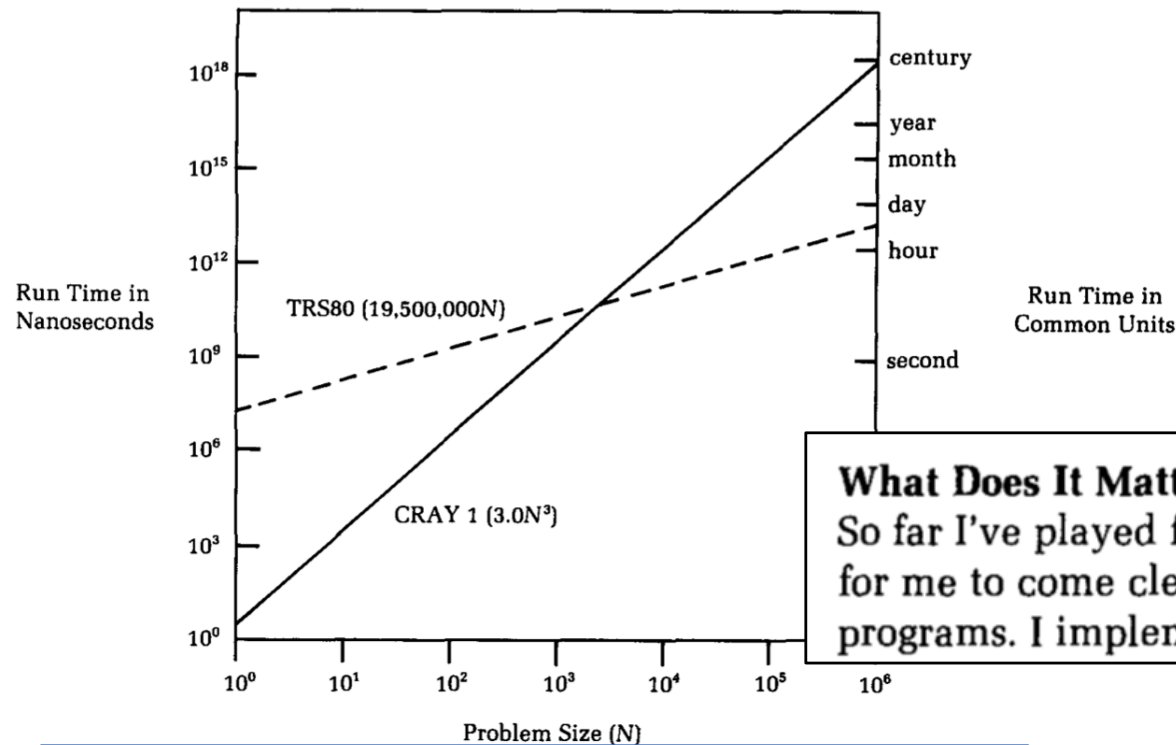
The history of the problem sheds light on the algorithm design techniques. The problem arose in a pattern-matching procedure designed by Ulf Grenander of Brown University in the two-dimensional form described in Problem 7. In that form, the maximum sum subarray was the maximum likelihood estimator of a certain kind of pattern in a digitized picture. Because the two-dimensional problem required too much time to solve, Grenander simplified it to one dimension to gain insight into its structure.

Grenander observed that the cubic time of Algorithm 1 was prohibitively slow, and derived Algorithm 2. In 1977 he described the problem to Michael Shamos of UNILOGIC, Ltd. (then of Carnegie-Mellon University)

who overnight designed Algorithm 3. When Shamos showed me the problem shortly thereafter, we thought that it was probably the best possible; researchers had just shown that several similar problems require time proportional to $N \log N$. A few days later Shamos described the problem and its history at a seminar attended by Jay Kadane (a statistician at Carnegie-Mellon University), who designed the linear-time Algorithm 4 within a minute. Fortunately, we know that there can be no better algorithm: any algorithm must look at all N inputs.

Even though the one-dimensional problem is now completely solved, Grenander's original two-dimensional problem remains open. Because of the computational expense of all known algorithms, Grenander had to abandon that approach to the pattern-matching problem. Readers who feel that the linear-time algorithm for the one-dimensional problem is "obvious" are therefore urged to find an "obvious" algorithm for Problem 7!

...e il suo significato computazionale



What Does It Matter?

So far I've played fast and loose with "big-ohs"; it's time for me to come clean and tell about the run times of the programs. I implemented the four primary algorithms

ter much. To underscore this point, I conducted an experiment in which I tried to make the constant factors of two algorithms differ by as much as possible. To

The Tyranny of Asymptotics

N	Cray-1, FORTRAN, Cubic Algorithm	TRS-80, BASIC, Linear Algorithm
10	3.0 microsecs	200 millisecs
100	3.0 millisecs	2.0 secs
1000	3.0 secs	20 secs
10,000	49 mins	3.2 mins
100,000	35 days	32 mins
1,000,000	95 yrs	5.4 hrs

...e il suo significato computazionale

TABLE I. Summary of the Algorithms

Algorithm		1	2	3	4
Lines of C Code		8	7	14	7
Run time in microseconds		$3.4N^3$	$13N^2$	$46N \log N$	$33N$
Time to solve problem of size	10^2	3.4 secs	130 msecs	30 msecs	3.3 msecs
	10^3	.94 hrs	13 secs	.45 secs	33 msecs
	10^4	39 days	22 mins	6.1 secs	.33 secs
	10^5	108 yrs	1.5 days	1.3 min	3.3 secs
	10^6	108 mill	5 mos	15 min	33 secs
Max problem solved in one	sec	67	280	2000	30,000
	min	260	2200	82,000	2,000,000
	hr	1000	17,000	3,500,000	120,000,000
	day	3000	81,000	73,000,000	2,800,000,000
If N multiplies by 10, time multiplies by		1000	100	10+	10
If time multiplies by 10, N multiplies by		2.15	3.16	10-	10